14 Tree-Based Learners

14.1 Introduction

In this chapter we greatly expound on our discussion of *tree-based* learners, first introduced in Section 11.2.3, which are wildly popular due to their great effectiveness particularly with structured data (see, e.g., the discussion in Section 11.8). In this chapter we explore the technical eccentricities associated with tree-based learners, describe the so-called *regression* and *classification trees*, and explain their particular usage with boosting based cross-validation and bagged ensembles (first introduced in Sections 11.5 and 11.9, respectively) where they are referred to as *gradient boosting* and *random forests* in the jargon of machine learning.

14.2 From Stumps to Deep Trees

In Section 11.2.3 we introduced the simplest exemplar of a tree-based learner: the stump. In this section we discuss how, using simple stumps, we can define more general and complex tree-based universal approximators.

14.2.1 The stump

The most basic tree-based universal approximator, the stump, is a simple step function of the form

$$f(x) = \begin{cases} v_1 & x \leq s \\ v_2 & x > s \end{cases} \tag{14.1}$$

with three tunable parameters: two step levels or *leaf* parameters denoted by v_1 and v_2 (whose values are set independently of one another), and a split point parameter s defining the boundary between the two levels. This simple stump is depicted in the top-left panel of Figure 14.1. In the top-right panel of this figure we show another graphical representation of the generic stump in Equation (14.1), which helps explain the particular nomenclature (i.e., trees, leaves, etc.) used in the context of tree-based approximators. Represented this

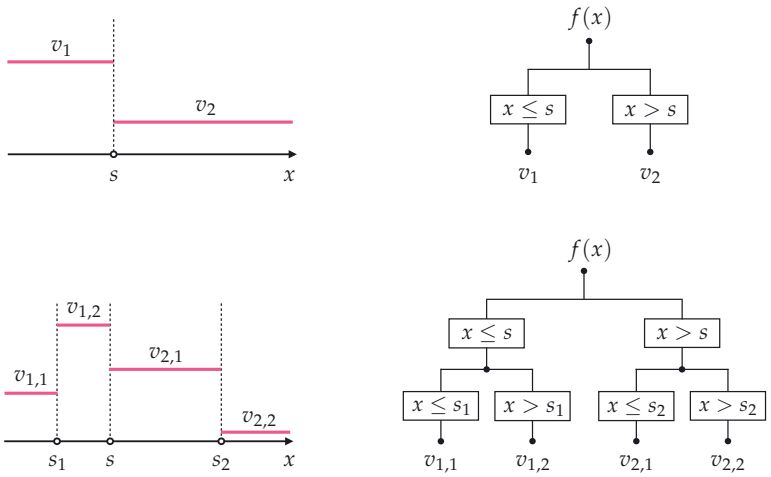

Figure 14.1 (top-left panel) A simple stump, defined in Equation (14.1). (top-right panel) A graphical illustration of a stump function as a binary tree. (bottom-left panel) A depth-two tree formed by recursing on each leaf of the stump, replacing it with a new stump. (bottom-right panel) A graphical illustration of the depth-two tree.

way, the stump can be thought of as a binary tree structure of *depth one*, with $f(x)$ as its root node, and v_1 and v_2 as its leaf nodes.

The stump defined in Equation (14.1) takes in one-dimensional (i.e., scalar) input. When the input is N-dimensional in general, the stump cuts along a single dimension (or coordinate axis). For example, when defined along some nth dimension a stump taking in N-dimensional input $\mathbf{x}$ is defined as

$$f(\mathbf{x}) = \begin{cases} v_1 & x_n \leq s \\ v_2 & x_n > s \end{cases} \tag{14.2}$$

where x_n here denotes the nth dimension of $\mathbf{x}$.

14.2.2 Creating deep trees via recursion

Recursing we can construct deeper trees by applying the same concept used to build a stump to each of its leaves, i.e., by splitting each leaf in two. This recursion results in a tree of *depth two*, with three split points and four distinct leaves, as shown in the bottom row of Figure 14.1. A depth-two tree has significantly greater capacity (see Section 11.2) than a stump, since the location of the split points and the leaf values can be set in a multitude of different ways, as can be seen in the top row of Figure 14.2.

This recursive idea can then be continually applied to each leaf of a depth-two tree to create a depth-three tree, and so forth. The deeper a tree becomes the more capacity it gains, with each unit being able to take on a wider variety

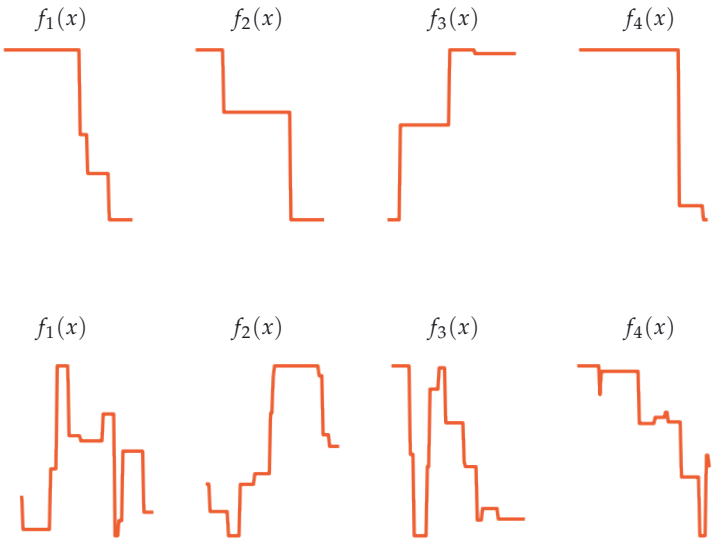

Figure 14.2 Four instances of a depth $D = 2$ tree (top row) and a depth $D = 10$ tree (bottom row), where in each instance all the parameters (i.e., split points and leaf values) are set at random. The latter is clearly capable of generating a wider swath of shapes given different settings of its parameters, and thus has higher capacity. Note here that the leaves are connected by vertical lines in order to give each tree instance a continuous appearance, which is done for visualization purposes only.

of different shapes, as can be seen in the bottom row of Figure 14.2. Indeed this is reflected in the fact that trees become exponentially more parameterized the deeper they are made: one can easily show that a tree of general depth D (with scalar input) will have $2^D - 1$ split points and 2^D leaves, thus $2^{D+1} - 1$ tunable parameters in total. This recursive procedure is often referred to as the *growing of a tree* in the jargon of machine learning.

Note importantly that *unlike* fixed-shape and neural network universal approximators, tree-based units are defined *locally*. This means that when we adjust one parameter of a polynomial or a neural network unit it can *globally* affect the shape of the function over the entire input space. However, when we split any leaf of a tree we are only affecting the shape of the tree locally at that leaf. This is why tree-based universal approximators are sometimes called *local function approximators*.

14.2.3 Creating deep trees via addition

Deeper, more flexible trees can also be constructed via *addition* of shallower trees. For instance, Figure 14.3 illustrates how a depth-two tree can be created by adding together three depth-one trees (i.e., three stumps). Again, it is easy

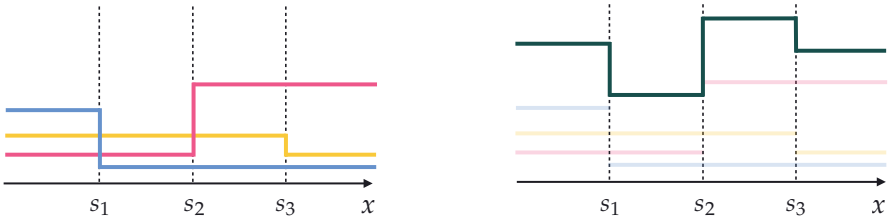

Figure 14.3 (left panel) Three stumps, each depicted in a distinct color. For ease of visualization we have connected the leaves by vertical lines in order to give each stump instance a continuous appearance. (right panel) A depth-two tree (in black) created by adding together the three stumps shown in the left panel.

to show that in general adding $2^D - 1$ stumps (with scalar input) together will create a depth-D tree (provided that the stumps do not share any split points).

14.2.4 Human interpretability

Given their particularly simple structure, shallow tree-based units (such as the depth-one and depth-two trees shown in Figure 14.1) are often easy to interpret by humans, in comparison to their fixed-shape and neural network counterparts. However, this feature of tree-based units quickly dissipates as the depth of a tree is increased (see, e.g., depth $D = 10$ trees depicted in the bottom row of Figure 14.2) as well as when trees are combined or *ensembled* together.

14.3 Regression Trees

In this section we discuss the use of general tree-based universal approximators for the problem of regression, often called *regression trees*. Unlike fixed-shape or neural network universal approximators, cost functions imbued with tree-based units create highly nonconvex, staircase-like functions that cannot be easily minimized by any local optimization method. To see why this is the case via an example, let us take the simple regression dataset shown in the top-left panel of Figure 14.4, and try to fit a nonlinear regressor to it using a model composed of a single stump, by minimizing an appropriate cost function over this model (e.g., the Least Squares cost).

14.3.1 Determining an optimal split point when leaf values are fixed

Fitting a single-stump model to our dataset entails tuning its three parameters: the location of the stump's split point, as well as its two leaf values. To make matters easy, here we fix the two leaf parameters associated with our model to two arbitrary values so that only the split point parameter s remains to be

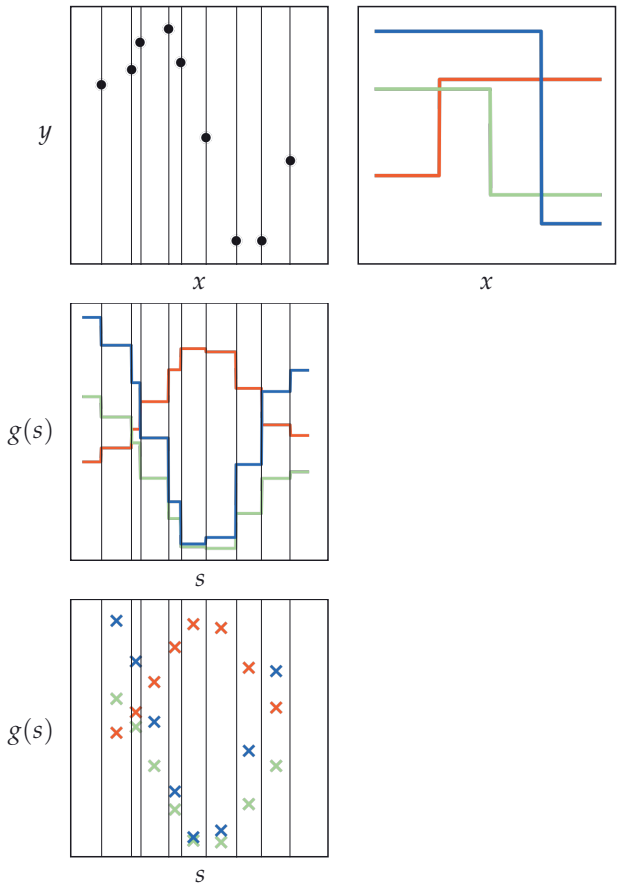

Figure 14.4 (top-left panel) A prototypical nonlinear regression dataset. (top-right panel) Three stumps with fixed leaf values (whose split point can vary). (middle panel) Each stump instance is slid horizontally across the input of the data by varying its split point value, creating three corresponding staircase-like Least Squares costs. (bottom panel) Each cost in the middle panel is constant in between consecutive inputs, implying that we need only test one split point per flat region, e.g., the mid-point, as shown in this panel. See text for further details.

optimally tuned, and hence we can now visualize the *one-dimensional* Least Squares cost function $g(s)$ involving the split point parameter alone. In the top-right panel of Figure 14.4 we show three stump instances, colored red, green, and blue, with distinct but fixed leaf values. Now we take each stump and sweep it over the dataset horizontally, trying out for each all possible split points in the input space of the dataset. The three Least Squares cost functions resulting from this exercise are shown in the middle panel of the figure and are colored to match their corresponding stumps shown in the top-right panel. Each one-dimensional cost, as we can see, looks like a staircase consisting of many

perfectly flat regions. These problematic flat regions are a direct consequence of the shape of our nonlinearity (i.e., the stump). Recall, we saw similar behavior when dealing with step functions in the context of logistic regression back in Sections 6.2 and 6.3. The existence of such flat regions is massively undesirable because no local optimization algorithm can navigate them effectively.

However, notice when the leaf values of a stump are fixed the corresponding Least Squares cost remains constant for all split point values *in between* consecutive inputs. In other words, all three cost functions in the middle panel of Figure 14.4 take on a staircase shape with their flat step areas located in the same locations: the regions in between consecutive input values.

This fact has a very practical repercussion: while we cannot properly tune the split point parameter using local optimization (due to the staircase shape of the cost function over this parameter), we can, however, find one by simply testing a *single value* (e.g., the mid-point) in each of the flat areas of the cost since all split points there produce the same regression quality. This collection of mid-point evaluations for each of our three example stumps is illustrated in the bottom panel of Figure 14.4.

14.3.2 Determining optimal leaf values when split point is fixed

Contrary to the task of determining the optimal split point of a stump with fixed leaf values, determining the optimal leaf values for a stump with a fixed split point is exceedingly straightforward. Since the leaves of a stump are *constant-valued*, and we want them both to be set so that together our stump represents the data as well as possible, it makes intuitive sense simply to set the value of each leaf to the *mean output* of those points it will represent. This choice is shown in red in the right panel of Figure 14.5 for our toy regression dataset, where for a given split point (illustrated by the vertical blue dashed line in the left panel) the leaf value on the *left* is set to the mean of the output of those points lying to the *left* of the split point, and the leaf value on the *right* is set to the mean of the output of those points to the *right* of the split point.

This intuitive choice of leaf value can be completely justified via the *first-order optimality condition* introduced in Section 3.2. To see this, let us first formalize the general scenario we are investigating. In the context of regression, our fixed split point s is defined along the nth input dimension of a regression dataset – denoted by $\left\{\left(\mathbf{x}_p, y_p\right)\right\}_{p=1}^{P}$ – and splits the data into two sections. We can keep track of these two subsets of our data via index sets Ω_L and Ω_R, which denote the input/output pairs of our dataset lying on either side of the split to the "left" and "right" of it, expressed formally as

$$\Omega_L = \left\{p \,|\, x_{p,n} \leq s\right\} \qquad \text{and} \qquad \Omega_R = \left\{p \,|\, x_{p,n} > s\right\}. \tag{14.3}$$

A general stump using this split point (echoing Equation (14.2)) can then be written as

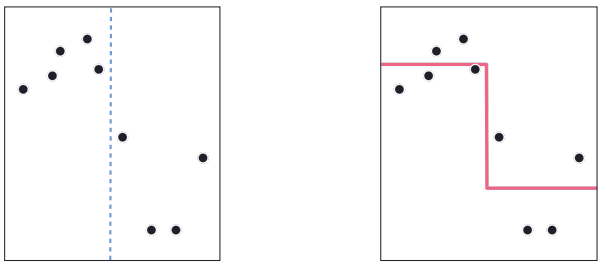

Figure 14.5 (left panel) The same regression dataset shown in Figure 14.4, along with the fixed split point shown via a vertical dashed blue line that divides the input space of data into two subspaces lying to the left and right of this line. (right panel) The stump with optimally set leaf values, determined as the mean of the output of all data points to the left and right of the split point. See text for further details.

$$f(\mathbf{x}) = \begin{cases} v_L & x_n \leq s \\ v_R & x_n > s \end{cases} \tag{14.4}$$

where x_n is the nth dimension of the input $\mathbf{x}$, and v_L and v_R are leaf values we will determine.

To determine the optimal values of v_L and v_R we can minimize two *one-dimensional* Least Squares costs, defined over the points belonging to the index set Ω_L and Ω_R, respectively, as

$$g(v_L) = \frac{1}{|\Omega_L|}\sum_{p\in\Omega_L}\left(v_L - y_p\right)^2 \qquad \text{and} \qquad g(v_R) = \frac{1}{|\Omega_R|}\sum_{p\in\Omega_R}\left(v_R - y_p\right)^2 \tag{14.5}$$

where $|\Omega_L|$ and $|\Omega_R|$ denote the number of points belonging to the index sets Ω_L and Ω_R, respectively.

Each of these cost functions is exceptionally simple. Setting the derivative of each to zero (with respect to its corresponding leaf value) and solving gives the optimal leaf values $v_L^\star$ and $v_R^\star$, respectively, as

$$v_L^\star = \frac{1}{|\Omega_L|}\sum_{p\in\Omega_L} y_p \qquad \text{and} \qquad v_R^\star = \frac{1}{|\Omega_R|}\sum_{p\in\Omega_R} y_p. \tag{14.6}$$

14.3.3 Optimization of regression stumps

Combining the two ideas discussed previously provides a reasonable work-around for tuning all three parameters of a stump for the purposes of regression (as an alternative to tuning all three together via local optimization, which we cannot do). That is, first we create a set of candidate split point values by recording every mid-point between our input data, along each of its input

dimensions. For each candidate split point we then determine the stump's leaf values optimally, setting them to the mean of the training data output to the left and right of the split point, and compute its (Least Squares) cost value. After doing this for all candidate split points, we find the very best stump (with optimal split point and leaf values) as one that provides the lowest cost value.

Example 14.1 Fitting the parameters of a simple regression tree

In this example we fit a single-stump model to the toy regression dataset shown in Figures 14.4 and 14.5, illustrating the entire range of candidate stumps whose split points are formed by taking the mid-point between each consecutive pair of inputs and whose corresponding leaf values are set to the mean of the output on either side of each split. Scanning the panels of Figure 14.6 from the top-left to the bottom-right we illustrate the entire range of candidate stumps for this dataset, scanning from left to right across the input of the dataset. In the top of each panel we show the candidate stump, with the Least Squares cost values associated with that particular stump as well as those that preceded it plotted underneath. Once all candidates have been tested (as shown in the bottom-right panel), the particular stump providing the lowest possible cost value (here, the fifth stump in the bottom-left panel) is found optimal.

In general, for a dataset of P points, each of input dimension N, there are a total of $N(P-1)$ split points to choose from over the entire input space of the problem. When computation becomes a primary concern (mainly with very large P and/or N) sampling strategies derived from this basic scheme may be used, including testing split points along only a random selection of all input dimensions, testing a coarser selection of split points, and so on.

14.3.4 Deeper regression trees

To fit a depth-two tree to a regression dataset we first fit a stump as described in the previous section, and then recurse on the same idea on each of the stump's leaves. In other words, we can think of the first fitted stump as dividing our original dataset into two nonoverlapping subsets, one belonging to each leaf. Thinking recursively we can then fit a stump to each of these subsets in precisely the same way as we fit the stump to the original dataset, splitting each of the leaves of our original stump in two and creating a depth-two tree. We can go on further and repeat this process, splitting each leaf of our depth-two tree to create a depth-three tree, and so on.

Note that in the process of *growing* the tree, there are certain conditions in which we should *not* split a leaf. For example, there is no reason to split a leaf that contains just a single data point, or one where data points contained in

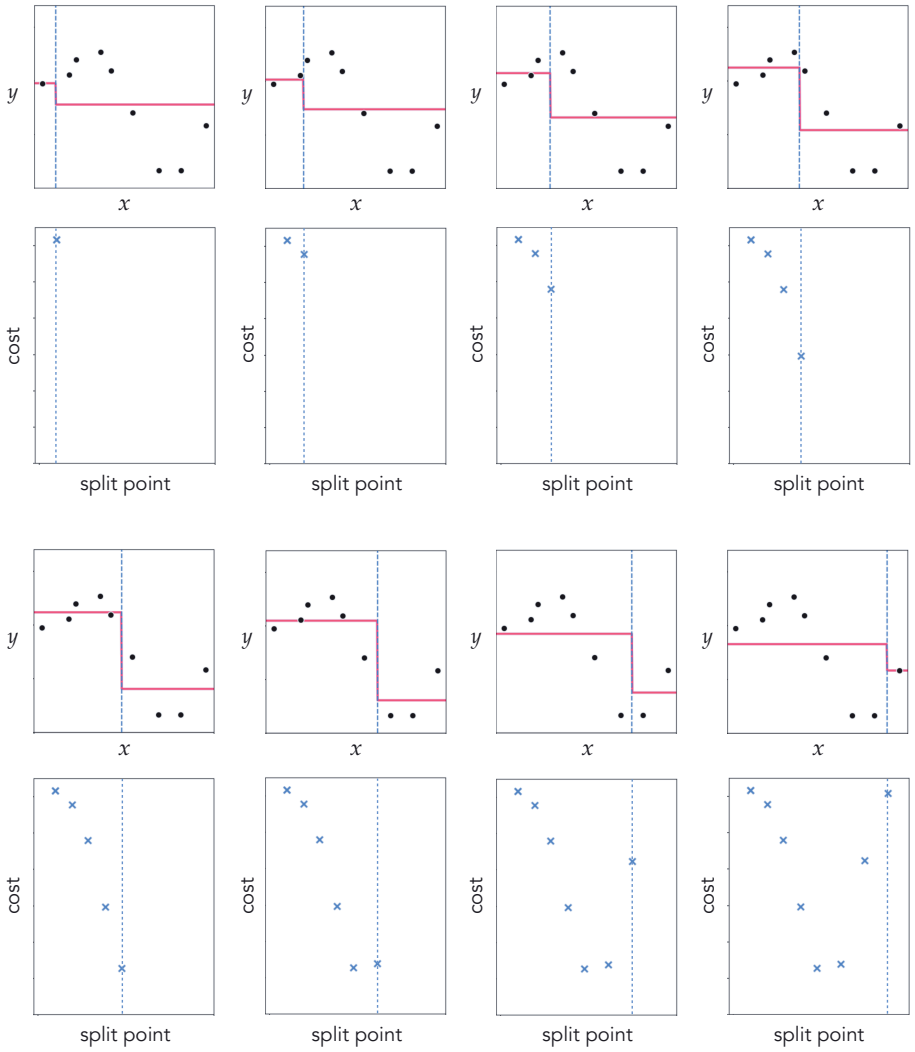

Figure 14.6 Figure associated with Example 14.1. See text for details.

the input space of the leaf have precisely the same output value (since in both instances our current leaf represents the data contained in it perfectly). Both of these practical conditions are often generalized when implementing recursive tree-based regressors in practice. For instance, progress on a leaf may be halted if it contains less than a predetermined number of points (as opposed to just a singleton point). Thus, in practice, a regression tree (with scalar input) of depth D may not end up with precisely 2^D leaves. Instead, certain branches stemming from the root of the tree may halt sooner than others, with some branches of the tree possibly growing to the defined depth. Therefore when applying binary

trees to regression (and, as we will see, classification) we refer to the trees as having a *maximum depth*, i.e., the largest depth that a branch of the tree can possibly grow to.

Example 14.2 Growing a maximum-depth regression tree

The recursive procedure for growing a deep regression tree is illustrated in Figure 14.7. We begin (on the left) by fitting a stump to the original dataset. As we move from left to right the recursion proceeds, with each leaf of the preceding tree split in order to create the next, deeper tree. As can be seen in the rightmost panel, a tree with maximum depth of four is capable of representing the training data perfectly.

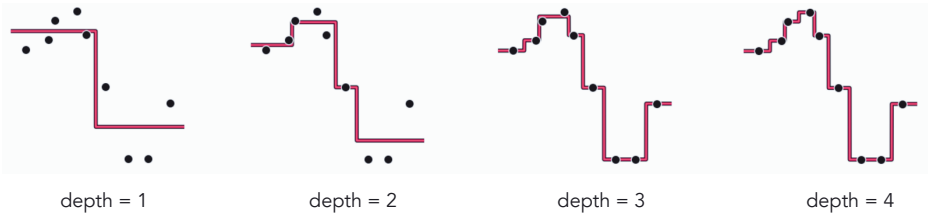

Figure 14.7 Figure associated with Example 14.2. See text for details.

14.4 Classification Trees

In this section we discuss the application of tree-based universal approximators to the problem of classification, often referred to as *classification trees*. Thankfully, virtually everything that we have previously seen regarding regression trees in the previous section caries over directly to the problem of classification. However, as we will see, the fact that classification data has *discrete* output naturally provokes different approaches to determining appropriate leaf values. As in the prior section, we begin here by discussing the proper construction of a stump, employing a toy dataset to illustrate key concepts, and recurse on this idea to create deeper trees.

14.4.1 Determining an optimal split point when leaf values are fixed

Imagine we are now dealing with a classification dataset, for example, the toy dataset shown in the left panel of Figure 14.8, and suppose that we aim to properly fit a stump to this data. If we attempt to set the *split point* of our stump via local optimization we run into precisely the same problem we came upon in Section 14.3.1 with regression. That is, not only will any corresponding

classification cost function be nonconvex, but it will consist of completely flat, staircase-like sections that no local optimization algorithm can navigate effectively. Thus determining the optimal split point value in the case of classification must naturally result in the same approach we saw with regression in the previous section: we must test out an array of split point candidates to determine which works best. Once again, for the same practical purposes we saw with regression, we can simply test the mid-points between each consecutive pair of input values along each of their input dimensions (or a subset of these points if their number becomes prohibitively large).

14.4.2 Determining optimal leaf values when split point is fixed

In Section 14.3.2 we saw with regression that the leaf values for a single-stump model can be intuitively set to the *mean* output of those points belonging to each leaf of the stump. We backed up this intuitive choice by showing that these settings are precisely what we find by solving the first-order condition for a set of appropriately defined Least Squares cost functions. Here in the classification scenario, let us follow this logic in reverse and begin by optimizing an appropriate cost function (e.g., the Perceptron or Cross Entropy/Softmax costs). We then follow by presenting an intuitive choice based on a different statistic of the output: the *mode*.

Suppose we are tasked with classification of a two-class dataset $\left\{\left(\mathbf{x}_p, y_p\right)\right\}_{p=1}^{P}$ with label values $y_p \in \{-1, +1\}$, and that the split point of our stump is fixed. We define index sets Ω_L and Ω_R as in Equation (14.3) to denote the indices of all points lying to the "left" and "right" of our split point. To determine the optimal values of v_L and v_R we can minimize two *one-dimensional* classification costs (e.g., the Softmax) defined over the points belonging to the index set Ω_L and Ω_R, respectively, as

$$g(v_L) = \frac{1}{|\Omega_L|}\sum_{p\in\Omega_L}\log\left(1 + e^{-y_p v_L}\right) \qquad \text{and} \qquad g(v_R) = \frac{1}{|\Omega_R|}\sum_{p\in\Omega_R}\log\left(1 + e^{-y_p v_R}\right) \tag{14.7}$$

where again, as in Equation (14.5), $|\Omega_L|$ and $|\Omega_R|$ denote the number of points belonging to the index sets Ω_L and Ω_R, respectively. One can also weight the summands of such cost functions (as detailed in Section 6.9.3) in order to better deal with potential *class imbalance* in the leaves.

In either case, unlike the analogous pair of Least Squares costs in Equation (14.5), here we cannot solve the corresponding first-order conditions in closed form and must rely on local optimization techniques. However, because of the simplicity of each problem such optimizations are especially easy to solve iteratively. Indeed, often these sorts of costs are approximately minimized by applying just a *single* step of Newton's method. Doing this substantially mini-

mizes the costs while keeping computation overhead low[1] and preventing potential numerical issues associated with the Softmax cost and Newton's method (introduced in the context of linear two-class classification in Section 6.6).

Note that, as with any other approach to classification, once appropriate leaf values have been determined, to make valid predictions the output of a classification stump must be passed through an appropriate discretizer, e.g., the sign function in the case of two-class classification using label values ± 1 (see Section 6.8.1).

As an alternative to the *cost function* based approach detailed thus far, one can also choose optimal leaf values based on simple statistics of the output. Since the output of classification data is *discrete*, we would naturally avoid using the *mean* as our statistic of choice (as we did with regression), and instead lean towards using the *mode* (i.e., the most popular output label), also called the *majority vote*. Using the mode will keep our leaf values constrained to the discrete labels of our data, providing more appropriate stumps.

However, the standard mode statistic can lead to undesirable consequences, an example of which is illustrated for a simple two-class dataset in Figure 14.8. For this particular dataset because of the distribution of the majority class (here those points with label value $y_p = -1$) the statistical mode on *both* sides of *every* stump will always equal -1, and thus all stumps will be entirely flat and identical. The lack of stump diversity in this simple example does not invalidate the use of the standard mode. However, it does highlight its inefficiency in that deeper trees (which are more costly to create) are needed to capture the nonlinearity of such a toy dataset.

To compensate for class imbalances like the one shown here we can, in complete analogy to the concept of weighting cost functions to better handle class imbalance (see Section 6.8.1), choose leaf values based on the *balanced mode* or *balanced majority vote*. To compute the standard mode on one leaf of a stump we simply count up the number of points belonging to each class in the leaf, and determine the mode by picking the class associated with the largest count. To compute the *balanced* mode on one leaf of a stump we first count up the number of points belonging to each class on the leaf and then weight each count inversely based on the number of points in each class belonging to both leaves of the stump, determining the balanced mode by choosing the largest resulting weighted count. For a general multi-class dataset with C classes, the weighted count for the cth class on one leaf can be written as

[1] As detailed in Section 4.3, a single Newton step involves minimizing the best *quadratic approximation* to a cost function provided by its second-order Taylor series expansion, and for a general single-input cost $g(w)$ results in a simple update of the form

$$w^\star = w^0 - \frac{\frac{\mathrm{d}}{\mathrm{d}w} g(w^0)}{\frac{\mathrm{d}^2}{\mathrm{d}w^2} g(w^0) + \lambda} \tag{14.8}$$

where w^0 is some initial point, $\lambda \geq 0$ is a regularization parameter used to prevent possible division by zero (as discussed in Section 4.3.3), and $w^\star$ is the optimal update.

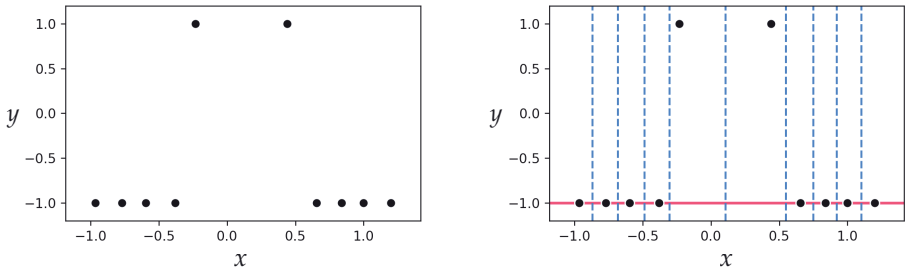

Figure 14.8 (left panel) A simple two-class classification dataset. (right panel) For each of the nine split points denoted by vertical dashed blue lines, assigning leaf values based on the *standard* mode leads to a completely flat stump, shown here in red. See text for further details.

$$\frac{\text{number of points from class } c \text{ in leaf}}{\text{number of points from class } c \text{ in both leaves of the stump}}. \tag{14.9}$$

Figure 14.9 shows the result of using this strategy (of setting the leaf values based on balanced mode instead of the mode itself) on the same dataset illustrated previously in Figure 14.8. Using the balanced mode here we produce a greater variety of stumps (when compared with using the standard mode), which allows us to capture the nonlinearity present in this dataset more effectively. To see how the balanced majority was used to define the leaf values in this instance let us examine one of the stumps (the sixth one in the middle-right panel) more closely. This stump has six data points on its left and four data points on its right side. Of the six data points lying to the left of its split point, four points have a label value of -1 and two points a label value of $+1$, resulting in a balanced majority vote of $\frac{4}{8}$ and $\frac{2}{2}$ for the two classes, respectively (noting that in this dataset there are a total of eight data points in the -1 class and two in the $+1$ class). Since $\frac{2}{2} > \frac{4}{8}$ the leaf value on the left is set to $+1$. Likewise, the balanced majority votes to the right of the split point for the -1 and $+1$ classes are calculated similarly as $\frac{4}{8}$ and $\frac{0}{2}$, respectively, and hence the right leaf value is set to -1.

14.4.3 Optimization of classification stumps

Putting everything together, to determine an optimal stump (consisting of optimal split point and leaf values) we can range over a set of reasonably chosen split points and construct corresponding leaf values for each stump using either the *cost function* based or *majority vote* based approaches described in Section 14.4.2. To determine which stump is ideal for our dataset we can then compute an appropriate classification metric over every stump instance and choose the one that provides the best performance. For example, with two-class classification we can employ an accuracy metric like those introduced in Section 6.8,

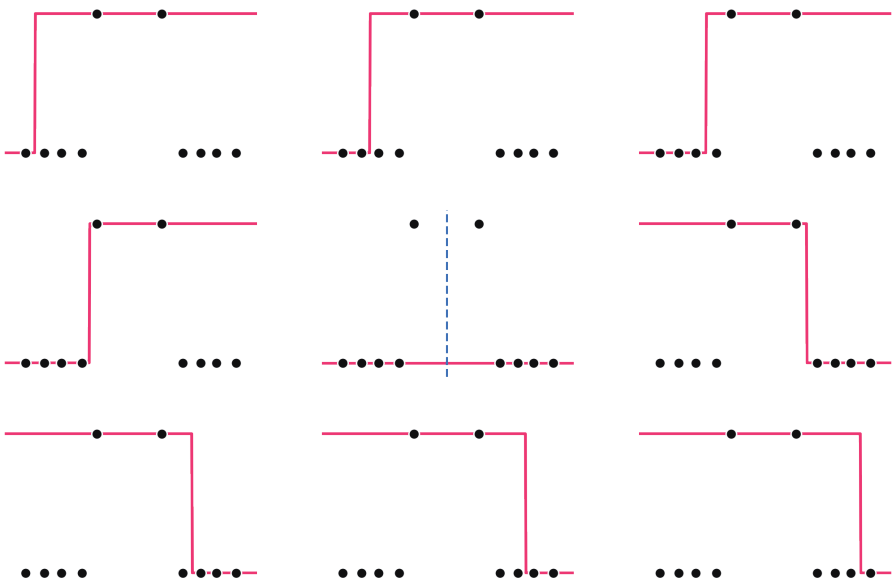

Figure 14.9 The same dataset and set of split points shown in Figure 14.8, only here the *balanced* mode calculation in Equation (14.9) is used to create the leaf values. See text for further details.

with the *balanced accuracy* discussed in Section 6.8.4 being the safest choice given the class imbalance we might encounter in practice, or more specialized metrics such as *information gain* [65].

Note that, in the context of classification trees, quality metrics are often referred to as *purity* metrics, since they measure how pure each leaf of the stump is in terms of class representation. Ideally, the stump is chosen that best represents the data while its leaves remain as "pure" as possible, each containing (largely) members of a single class if possible.

Example 14.3 Fitting the parameters of a simple classification tree

In Figure 14.10 we illustrate the resulting balanced accuracy of the stumps shown in Figure 14.9. Because of the symmetry of this particular dataset only the first five stumps are shown here, of which the fourth one provides the minimum cost, and thus is optimal for our dataset.

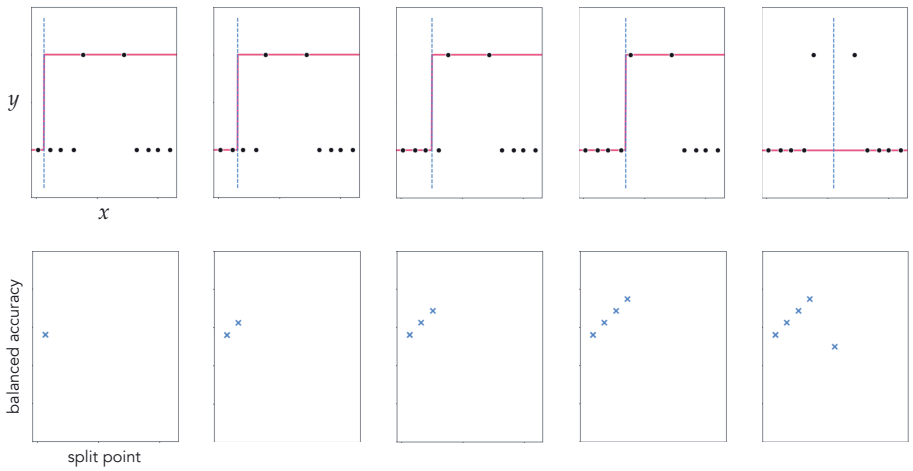

Figure 14.10 Figure associated with Example 14.3. See text for details.

14.4.4 Deeper classification trees

To build deeper classification trees we recurse on the two leaves of a stump and treat each like we did the original data, building a stump out of each. Just as with regression trees (see Section 14.3.4) this often results in binary trees of a *maximum depth*, as certain branches halt under obvious and/or user-defined conditions. With classification, one natural halting condition is when a leaf is completely *pure*, that is, it contains only members of a single class. In such a case there is no reason to continue splitting such a leaf. Other common halting conditions often used in practice include halting growth when the number of points on a leaf falls below a certain threshold and/or when splits do not sufficiently increase accuracy.

Example 14.4 Growing two maximum-depth classification trees

In Figure 14.11 we illustrate the growth of a tree to a maximum depth of seven on a two-class classification dataset. In Figure 14.12 we do the same for a multi-class classification dataset with $C = 3$ classes. In both cases as the tree grows note how many parts of the input space do not change as leaves on the deeper branches become *pure*. By the time we reach a maximum depth of seven we have considerably overfit both datasets.

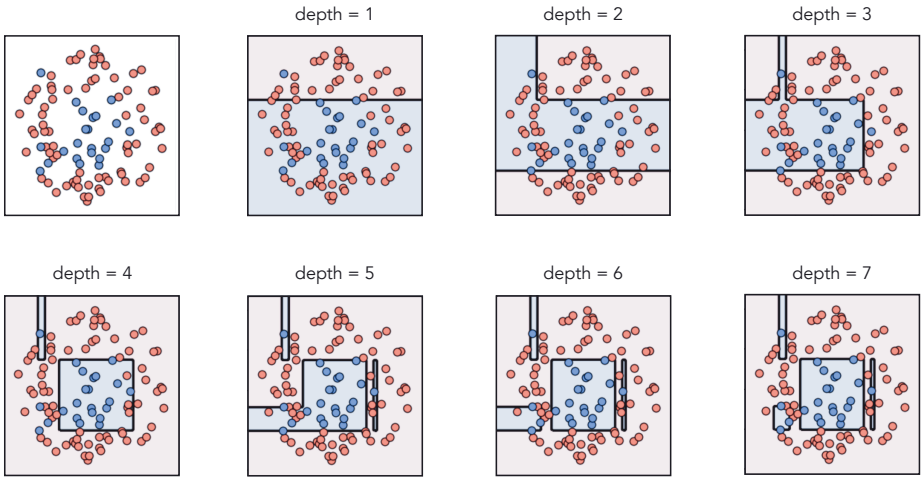

Figure 14.11 Figure associated with Example 14.4. See text for details.

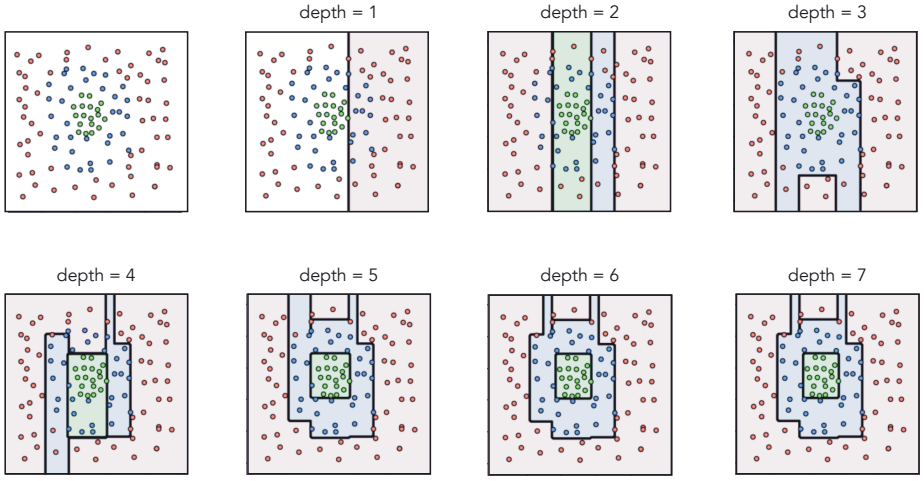

Figure 14.12 Figure associated with Example 14.4. See text for details.

14.5 Gradient Boosting

As mentioned in Section 14.2.3 deep tree-based universal approximators can be built via addition of shallow trees. The most popular way to build deeper regression and classification trees via addition is by summing together shallower ones, with each shallow tree constructed as detailed in Sections 14.3 and 14.4, and growing the tree sequentially one shallow member at a time. This scheme is an instance of the general *boosting* method introduced in Section 11.5. Moreover, trees are indeed the most popular universal approximator used when applying

boosting based cross-validation, with this pairing very often referred to as *gradient boosting* [66, 67]. The principles of boosting outlined in Section 11.5 remain unchanged in the context of tree-based learners. However, with the specific knowledge of how to appropriately fit regression and classification trees to data we can now expound on important details related to gradient boosting that we could not delve into previously.

14.5.1 Shallow trees rule

As described in Section 11.5.1 *low-capacity* units (of any universal approximator) are most often used with boosting in order to provide a fine-resolution model search. In the context of tree-based units this leads to the use of *shallow trees*, with stumps and trees of depth two being especially popular. One can of course use higher-capacity tree units (of depth three and beyond) in constructing deeper cross-validated trees via boosting. However, boosting with such high-capacity units can easily lead to skipping over of optimal models (as depicted abstractly in Figure 11.30).

14.5.2 Boosting with tree-based learners

As detailed in Section 11.5, at the mth round of boosting we begin with a model consisting of a fully tuned linear combination of $m-1$ units of a universal approximator (see Equation (11.26). In the case of tree-based learners we can dispense with the bias and weights of the linear combination (since they are naturally "baked in" to tree-based units, as detailed in Section 11.5.4) and write our model as

$$\text{model}_{m-1}\left(\mathbf{x}, \Theta_{m-1}\right) = f_{s_1}^{\star}\left(\mathbf{x}\right) + f_{s_2}^{\star}\left(\mathbf{x}\right) + \cdots + f_{s_{m-1}}^{\star}\left(\mathbf{x}\right) \tag{14.10}$$

where each function in this sum is a tree-based unit (e.g., a stump) whose split point(s) and leaf values have been chosen optimally. The mth round of boosting involves a search over a range of suitable candidates (here, various trees with differing split points) and a corresponding optimization of each candidate's leaf values. To construct the next candidate model we add a prospective unit $f_{s_m}\left(\mathbf{x}\right)$ to $\text{model}_{m-1}\left(\mathbf{x}, \Theta_{m-1}\right)$, forming

$$\text{model}_{m}\left(\mathbf{x}, \Theta_{m}\right) = \text{model}_{m-1}\left(\mathbf{x}, \Theta_{m-1}\right) + f_{s_m}\left(\mathbf{x}\right) \tag{14.11}$$

and optimize the leaf values of $f_{s_m}\left(\mathbf{x}\right)$ using an appropriate cost function (e.g., the Least Squares cost for regression and Softmax cost for classification) with respect to a training dataset. This leaf-value optimization very closely mirrors the approaches described in Sections 14.3.2 and 14.4.2 in the case of regression and classification, respectively.

For example, suppose f_{s_m} is a stump and we are dealing with the regression

case with a Least Squares cost and a dataset of P points denoted by $\left\{\left(\mathbf{x}_p, y_p\right)\right\}_{p=1}^{P}$. In complete analogy to Equation (14.5) we must then minimize the following pair of Least Squares costs

$$\begin{aligned} g(v_L) &= \frac{1}{|\Omega_L|}\sum_{p\in\Omega_L}\left(\text{model}_{m-1}\left(\mathbf{x}_p, \Theta_{m-1}\right) + v_L - y_p\right)^2 \\ g(v_R) &= \frac{1}{|\Omega_R|}\sum_{p\in\Omega_R}\left(\text{model}_{m-1}\left(\mathbf{x}_p, \Theta_{m-1}\right) + v_R - y_p\right)^2 \end{aligned} \tag{14.12}$$

to properly determine our two leaf values v_L and v_R, where Ω_L and Ω_R are index sets as defined in Equation (14.3), and $|\Omega_L|$ and $|\Omega_R|$ denote their sizes. Like those cost functions in Equation (14.5), these simple costs can each be minimized perfectly by checking the first-order condition for optimality (or equivalently by taking a single step of Newton's method).

Similarly, if dealing with two-class classification with a Softmax cost and label values $y_p \in \{-1, +1\}$ we set leaf values of a stump by minimizing two costs – analogous to Equation (14.7) – of the form

$$\begin{aligned} g(v_L) &= \frac{1}{|\Omega_L|}\sum_{p\in\Omega_L}\log\left(1 + e^{-y_p\left(\text{model}_{m-1}\left(\mathbf{x}_p, \Theta_{m-1}\right) + v_L\right)}\right) \\ g(v_R) &= \frac{1}{|\Omega_R|}\sum_{p\in\Omega_R}\log\left(1 + e^{-y_p\left(\text{model}_{m-1}\left(\mathbf{x}_p, \Theta_{m-1}\right) + v_R\right)}\right) \end{aligned} \tag{14.13}$$

which (in both cases) cannot be minimized in closed form, but must be solved via local optimization. Often, as discussed in Section 14.4.2, this is done by simply taking a single step of Newton's method as it provides a positive trade-off between the minimization quality and computation effort.

Example 14.5 Regression via gradient boosting

In Figure 14.13 we illustrate the use of boosting with regression stumps, which is often (as discussed in Section 11.5.6) interpreted as successive rounds of fitting to the *residual* of a regression dataset. We can see this in the case of a simple stump by rearranging terms in Equation (14.12). For example, $g(v_L)$ in Equation (14.12) can be rewritten as

$$g(v_L) = \frac{1}{|\Omega_L|}\sum_{p\in\Omega_L}\left(v_L - r_p\right)^2 \tag{14.14}$$

where r_p is the residual of the pth point defined as $r_p = y_p - \text{model}_{m-1}\left(\mathbf{x}_p, \Theta_{m-1}\right)$.

In the top row of this figure we show the original dataset along with the resulting fit provided by a model constructed from multiple rounds of stump-based boosting. Simultaneously in the bottom row we show each subsequent

stump-based fit to the residual provided by the most recent stump added to the running model.

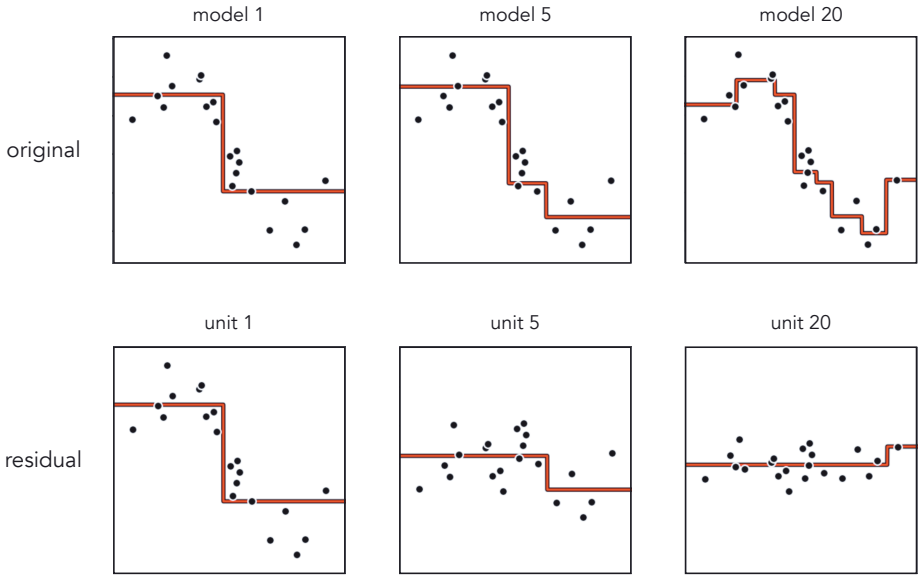

Figure 14.13 Figure associated with Example 14.5. See text for details.

Example 14.6 Spam detection via gradient boosting

In this example we use gradient boosting employing stumps, and cross-validate to determine an ideal number of rounds of boosting, using the spam dataset first described in Example 6.10. In this set of experiments we use the Softmax cost and set aside 20 percent of this two-class dataset (randomly) for validation purposes. We run 100 rounds of boosting and take a single step of Newton's method to tune each stump function. In Figure 14.14 we plot the number of misclassifications on both the training (in blue) and validation (in yellow) sets. The minimum number of misclassifications on the validation set occurred at the sixty-fifth round of boosting, which resulted in 220 and 50 misclassifications on the training and validation sets, respectively. A simple linear classifier trained on the same portion of data provided 277 and 67 misclassifications on training and validation sets, respectively.

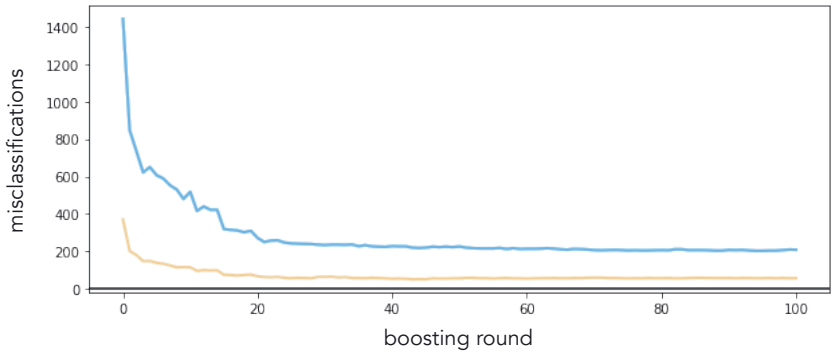

Figure 14.14 Figure associated with Example 14.6. See text for details.

14.6 Random Forests

Unless human interpretability of the final model is of primary concern, one virtually never uses a single recursively defined regression or classification tree, but a *bagged ensemble* of them. Generally speaking, bagging (as detailed in Section 11.9) involves combining multiple cross-validated models to produce a single higher-performing model. One can easily do this with recursively defined trees, employing the cross-validation techniques outlined in Section 14.7. However, in practice it is often unnecessary to grow each tree using cross-validation to temper their complexity. Instead, each tree can be trained on a random portion of training data taken from the original dataset and grown to a predetermined maximum depth, and afterwards bagged together.

This can be done with any universal approximator in principle but is especially practical with tree-based learners. This is both because trees are cheap to produce, and also because as *locally* defined approximators (see Section 14.2.2), it is natural to employ basic leaf-split halting protocols while growing the individual trees themselves (see Sections 14.3.4 and 14.4.4). While trees can certainly overfit, even when not cross-validated they are naturally prevented from exhibiting the sort of wild oscillatory overfitting behavior that is readily possible with fixed-shape or neural network models.[2] Thus bagging a set of overfitting trees can often successfully combat the sort of overfitting each tree presents, resulting in very effective models. Moreover, because each fully grown tree in such an ensemble can be learned efficiently, the computational trade-off, that is, training a large number of fully grown trees compared with a smaller number of cross-validated ones (each of which require more resources to construct), is often advantageous in practice.

Such an ensemble of recursively defined trees is often called a *random forest* [68] in the jargon of machine learning. The "random" part of the name *random forest*

[2] Compare, for instance, the overfitting behavior exhibited by each universal approximator in Figure 11.17.

refers both to the fact that each tree uses a random portion of the original data as training (which, by convention, is often sampled from the original dataset *with* replacement), and that often only a random subset of input feature dimensions are sampled for viable split points at each node in the trees produced. For each tree in such a forest often something like $\left\lfloor \sqrt{N} \right\rfloor$ of N features are chosen at random to determine split points.

Example 14.7 Random forest classification

In Figure 14.15 we show the result of bagging a set of five fully grown classification trees trained on different random portions of a simple two-class dataset (in each instance $\frac{2}{3}$ of the original dataset was used for training and the final $\frac{1}{3}$ was used for validation). Each of the five splits are illustrated in the small panels on the left along with the decision boundary provided by each trained model, with the validation data points in each case highlighted with a yellow boundary. Note that while most of the individual trees overfit the data, their ensemble (shown in the large panel on the right) does not. This ensembled model, as detailed in Section 11.9, is built by taking the *mode* of the five classification trees on the left.

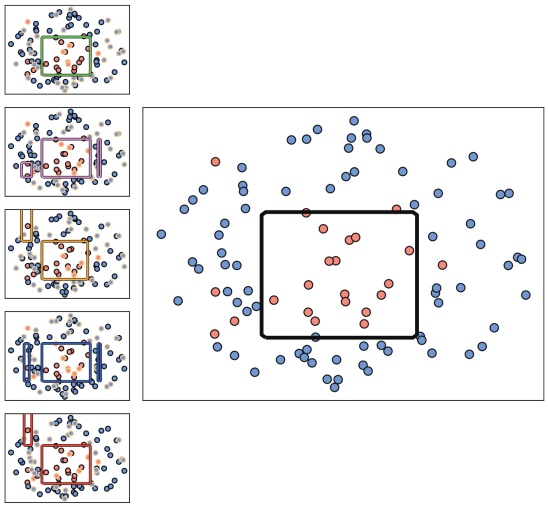

Figure 14.15 Figure associated with Example 14.7. (left column) The decision boundaries given by five fully grown classification trees, each grown on a different subset of the original data. Each individual tree tends to overfit the data but their bagged ensemble (shown in the right panel) compensates for this, and does not overfit. See text for further details.

14.7 Cross-Validation Techniques for Recursively Defined Trees

The basic principles of cross-validation, outlined in Sections 11.3 through 11.6, generally apply to the proper construction of recursively grown regression and classification trees in practice, with some technical differences arising due to the unique way such models are built. For example, we can begin with a low-capacity depth-one tree and grow it until minimum validation error is achieved (a form of *early stopping* specific to trees). Alternatively, we can begin by fitting a deep high-capacity tree to the data and gradually decrease its complexity by *pruning* leaves that do not contribute to low validation error (a form of *regularization* specific to trees).

Because recursively defined trees are typically ensembled as random forests, with each tree fully grown to a random training portion of the original dataset (as detailed in Section 14.6), the cross-validation techniques described here are often used to temper the complexity of a single regression or classification tree when *human interpretability* of a tree-based model is of crucial importance – something that is virtually always lost when ensembling multiple nonlinear models together.

14.7.1 Early stopping

We can easily use cross-validation to dictate the proper maximum depth of a tree by growing a tree of large depth, measuring validation error at each depth of the tree, and (after the fact) determine which depth produced minimal validation error. Alternatively, we can stop the growth early when we are confident[3] that (something approximating) minimum validation error has been achieved. This approach, while used in practice, translates to a relatively coarse model search since the capacity of a tree grows exponentially from one depth to the next.

As detailed in the previous two sections, practical considerations are often used to halt the leaf splitting (regardless of whether cross-validation is being performed). These include halting splitting if a leaf contains a singleton data point or a predecided (small) number of points, if all data points belong to the same class (in the case of classification) or have approximately the same output value (in the case of regression). To create a finer-resolution cross-validation search we can add validation-error-focused criteria to halt the growth of individual leaves as well. The simplest such criterion is to check whether or not splitting a leaf will result in *lowering* validation error (or lowering training error past a predetermined threshold): if yes, the leaf is split, otherwise growth of the leaf is halted. This approach to cross-validation is unique in that validation error

[3] As with any form of early stopping, determining when validation error is at its minimum "on the fly" is not always a straightforward affair as validation error does not always fall and rise monotonically (as discussed in the context of boosting and regularization in Sections 11.5.3 and 11.6.2, respectively).

will always monotonically decrease as the maximum depth of a tree is increased, but can result in *underfitting* models due to leaves halting growth prematurely.

Example 14.8 Early stopping by depth and leaf growth

In Figure 14.16 we show an example of cross-validating the maximum depth of a tree in the range of one through five for a simple regression dataset. The dataset, with training data colored in blue and validation data in yellow, is shown along with each subsequent fit in the top row of Figure 14.16, while the corresponding training/validation error is shown in its bottom row.

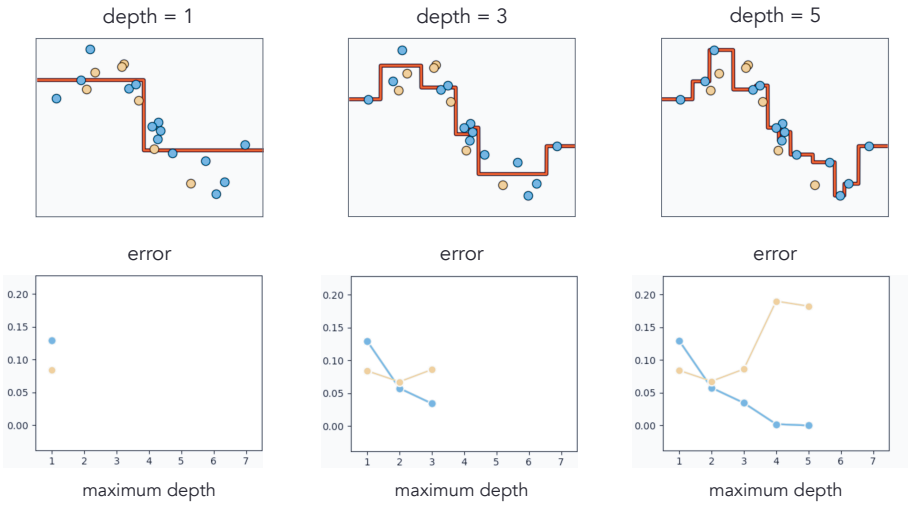

Figure 14.16 Figure associated with Example 14.8. See text for details.

In Figure 14.17 we show two examples of cross-validating the maximum depth of a tree in the range of one through five for another regression dataset, where leaf growth is now halted when validation error does not improve. In each instance the dataset, with training data colored in blue and validation data in yellow, is shown along with each subsequent fit on top, with the corresponding training/validation error is shown directly underneath it. In the first run (shown in the top two rows of Figure 14.17), because of the particular split of training and validation data, growth of each leaf is halted immediately, resulting in an underfitting, depth-one representation. With the second run on a different training–validation split of the data (shown in the bottom two rows of Figure 14.17), tree growth continues to improve validation error up to the maximum depth tested, resulting in a significantly better representation.

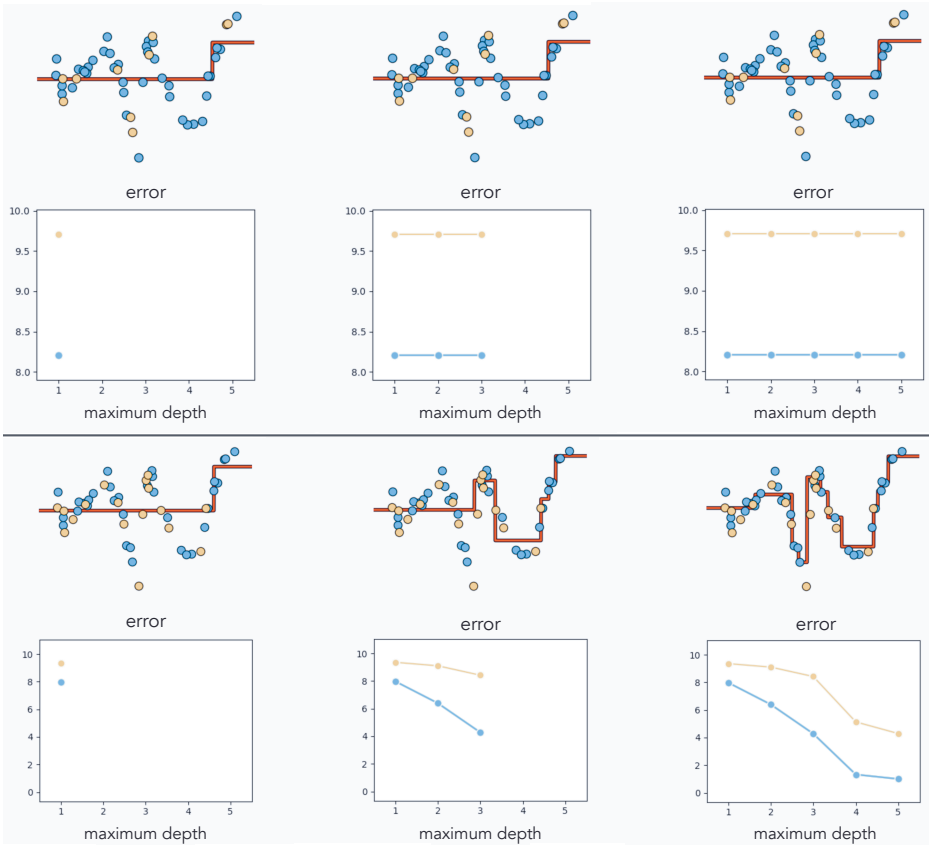

Figure 14.17 Figure associated with Example 14.8. See text for details.

14.7.2 Pruning

In contrast to beginning with a low-capacity (shallow) tree and growing it via early stopping, we can instead begin by fitting a high-capacity (deep) tree and remove leaves that do not improve validation error, until a minimum-validation tree structure remains. This technique – illustrated pictorially in Figure 14.18 – is called *pruning* because it entails examining an initially overly complicated tree and cutting off its leaves, akin to the way pruning of natural trees is done by snipping off redundant leaves and branches. Pruning is a tree-specific form of *regularization* based cross-validation, discussed previously in Section 11.6.

While early stopping is often more computationally efficient than pruning, the latter provides a finer-resolution model search in determining the tree structure with minimal validation error, since tree/leaf growth is unhindered and is only cut back after the fact.

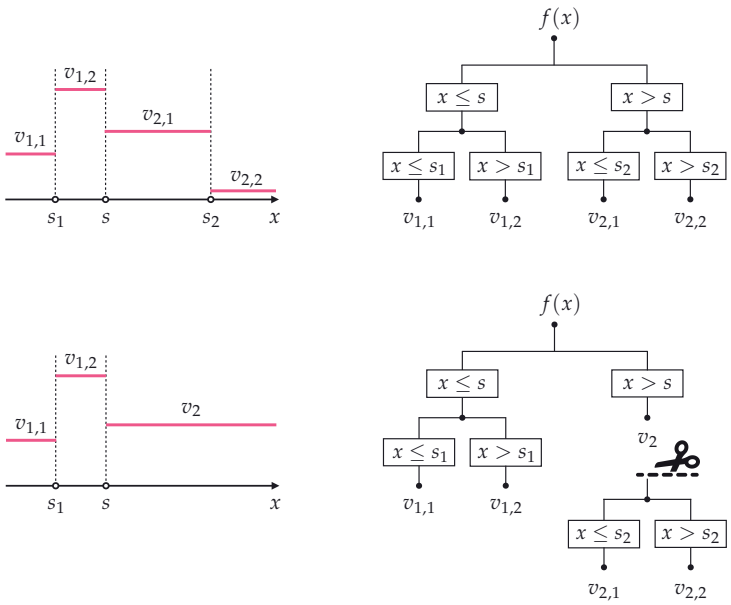

Figure 14.18 Pruning illustrated. (top panels) A fully grown tree of depth two with four leaves. (bottom) A pruned version of the original tree wherein the leaves $v_{2,1}$ and $v_{2,2}$ are pruned and replaced by a single leaf.

14.8 Conclusion

In this chapter we discussed a range of important technical matters related to tree-based universal approximators, which were first introduction in Section 11.2.3. We began in Section 14.2 by providing a more formal description of stumps as well as deeper trees, which as detailed in this section can be formed via recursion or summation. Recursively defined regression and classification trees were then gently motivated and detailed in Sections 14.3 and 14.4. Gradient boosting – the specific application of boosting based cross-validation (described in detail in Section 11.5) to tree-based learners – was touched upon in Section 14.5. Similarly, random forests – the specialized application of bagging (detailed in Section 11.9) to tree-based learners – was described in Section 14.6. Finally, the use of cross-validation with recursively defined tree-based learners – both from a naive and regularization perspective – were explored in Section 14.7.

14.9 Exercises

† The data required to complete the following exercises can be downloaded from the text's github repository at `github.com/jermwatt/machine_learning_refined`

14.1 Growing deep trees by addition
Show that in general adding $2^D - 1$ stumps (with scalar input) together will create a depth-D tree (provided that the stumps do not share any split points).

14.2 Fitting the parameters of a simple regression tree
Repeat the experiment described in Example 14.1, and reproduce the plots shown in Figure 14.6.

14.3 Code up a regression tree
Repeat the experiment described in Example 14.2 by coding up a recursively defined regression tree. You need not reproduce Figure 14.7. Instead, measure and plot the Least Squares error at each depth of your tree.

14.4 Code up a two-class classification tree
Repeat the first experiment described in Example 14.4 by coding up a recursively defined two-class classification tree. You need not reproduce Figure 14.11. Instead, measure and plot the number of misclassifications at each depth of your tree.

14.5 Code up a multi-class classification tree
Repeat the second experiment described in Example 14.4 by coding up a recursively defined multi-class classification tree. You need not reproduce Figure 14.12. Instead, measure and plot the number of misclassifications at each depth of your tree.

14.6 Gradient boosting for regression
Repeat the experiment described in Example 14.5 by coding up a gradient boosting algorithm employing regression stumps. Reproduce Figure 14.13, illustrating the boosted tree as well as the best stump fit to the residual at rounds one, two, and ten of boosting.

14.7 Gradient boosting for classification
Determine the leaf values of a stump added at the mth round of boosting to a classification tree by minimizing the Softmax costs in Equation (14.13) via taking a *single* step of Newton's method.

14.8 Random forests
Repeat the experiment described in Example 14.7 by coding up a random forest built from classification trees. You need not reproduce Figure 14.15. However, you can verify that your implementation is working properly by checking that

the final accuracy of your random forest classifier outstrips the accuracy of many of the individual trees in the ensemble or, alternatively, you can employ a testing set by setting aside a small portion of the original data.

14.9 Limitation of trees outside their training range

We have seen in this chapter that trees are efficient nonlinear approximations, and do not suffer from the sort of oscillatory behavior that can adversely affect global approximators like polynomials and neural networks (see Section 14.6). However, tree-based learners – by nature – fail to work effectively outside their training range. In this exercise you will see why this is the case by training a regression tree using the student debt data first shown in Figure 1.8. Use your trained tree to predict what the total student debt will be in the year 2050. Does it make sense? Explain why.

14.10 Naive cross-validation

Repeat the experiment outlined in Example 14.8 whose results are shown in Figure 14.16.